.NET Framework 4

**Implementing Finalize and Dispose to Clean Up Unmanaged Resources**

|  |
| --- |
| **Description: NoteNote** |
| For information about finalizing and disposing resources using C++, see [Destructors and Finalizers in Visual C++](http://msdn.microsoft.com/en-us/library/ms177197.aspx). |

Class instances often encapsulate control over resources that are not managed by the runtime, such as window handles (HWND), database connections, and so on. Therefore, you should provide both an explicit and an implicit way to free those resources. Provide implicit control by implementing the protected [Finalize](http://msdn.microsoft.com/en-us/library/system.object.finalize.aspx) on an object (destructor syntax in C# and C++). The garbage collector calls this method at some point after there are no longer any valid references to the object.

In some cases, you might want to provide programmers using an object with the ability to explicitly release these external resources before the garbage collector frees the object. If an external resource is scarce or expensive, better performance can be achieved if the programmer explicitly releases resources when they are no longer being used. To provide explicit control, implement the [Dispose](http://msdn.microsoft.com/en-us/library/system.idisposable.dispose.aspx) provided by the [IDisposable](http://msdn.microsoft.com/en-us/library/system.idisposable.aspx). The consumer of the object should call this method when it is finished using the object. **Dispose** can be called even if other references to the object are alive.

Note that even when you provide explicit control using **Dispose**, you should provide implicit cleanup using the **Finalize** method. **Finalize** provides a backup to prevent resources from permanently leaking if the programmer fails to call **Dispose**.

For more information about implementing **Finalize** and **Dispose** to clean up unmanaged resources, see [Garbage Collection](http://msdn.microsoft.com/en-us/library/0xy59wtx.aspx). The following example illustrates the basic design pattern for implementing **Dispose**. This example requires the [System](http://msdn.microsoft.com/en-us/library/system.aspx) namespace.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl08_code');" \o "Copy Code)

' Design pattern for a base class.

Public Class Base

Implements IDisposable

' Field to handle multiple calls to Dispose gracefully.

Dim disposed as Boolean = false

' Implement IDisposable.

Public Overloads Sub Dispose() Implements IDisposable.Dispose

Dispose(True)

GC.SuppressFinalize(Me)

End Sub

Protected Overloads Overridable Sub Dispose(disposing As Boolean)

If disposed = False Then

If disposing Then

' Free other state (managed objects).

disposed = True

End If

' Free your own state (unmanaged objects).

' Set large fields to null.

End If

End Sub

Protected Overrides Sub Finalize()

' Simply call Dispose(False).

Dispose (False)

End Sub

End Class

' Design pattern for a derived class.

Public Class Derived

Inherits Base

' Field to handle multiple calls to Dispose gracefully.

Dim disposed as Boolean = false

Protected Overloads Overrides Sub Dispose(disposing As Boolean)

If disposed = False Then

If disposing Then

' Release managed resources.

End If

' Release unmanaged resources.

' Set large fields to null.

disposed = True

End If

' Call Dispose on your base class.

Mybase.Dispose(disposing)

End Sub

' The derived class does not have a Finalize method

' or a Dispose method without parameters because it inherits

' them from the base class.

End Class

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl09_code');" \o "Copy Code)

// Design pattern for a base class.

public class Base: IDisposable

{

private bool disposed = false;

//Implement IDisposable.

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

// Free other state (managed objects).

}

// Free your own state (unmanaged objects).

// Set large fields to null.

disposed = true;

}

}

// Use C# destructor syntax for finalization code.

~Base()

{

// Simply call Dispose(false).

Dispose (false);

}

}

// Design pattern for a derived class.

public class Derived: Base

{

private bool disposed = false;

protected override void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

// Release managed resources.

}

// Release unmanaged resources.

// Set large fields to null.

// Call Dispose on your base class.

disposed = true;

}

base.Dispose(disposing);

}

// The derived class does not have a Finalize method

// or a Dispose method without parameters because it inherits

// them from the base class.

}

The following code expands the previous example to show the different ways **Dispose** is invoked and when **Finalize** is called. The stages of the disposing pattern are tracked with output to the console. The allocation and release of an unmanaged resource is handled in the derived class.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_code');" \o "Copy Code)

Imports System

Imports System.Collections.Generic

Imports System.Runtime.InteropServices

' Design pattern for a base class.

Public MustInherit Class Base

Implements IDisposable

Private disposed as Boolean = false

Private instName As String

Private trackingList As List(Of Object)

Public Sub New(instanceName As String, tracking As List(Of Object))

MyClass.instName = instanceName

trackingList = tracking

trackingList.Add(Me)

End Sub

Public ReadOnly Property InstanceName() As String

Get

Return instName

End Get

End Property

'Implement IDisposable.

Public Overloads Sub Dispose() Implements IDisposable.Dispose

Console.WriteLine(vbNewLine + "[{0}].Base.Dispose()", instName)

Dispose(true)

GC.SuppressFinalize(Me)

End Sub

Protected Overloads Overridable Sub Dispose(disposing As Boolean)

If disposed = False Then

If disposing Then

' Free other state (managed objects).

Console.WriteLine("[{0}].Base.Dispose(true)", instName)

trackingList.Remove(Me)

Console.WriteLine("[{0}] Removed from tracking list: {1:x16}",

instanceName, MyClass.GetHashCode())

Else

Console.WriteLine("[{0}].Base.Dispose(false)", instName)

End If

disposed = True

End If

End Sub

Protected Overrides Sub Finalize()

' Simply call Dispose(False).

Console.WriteLine(vbNewLine + "[{0}].Base.Finalize()", instName)

Dispose(False)

End Sub

End Class

' Design pattern for a derived class.

Public Class Derived

Inherits Base

Private disposed as Boolean = false

Private umResource As IntPtr

Public Sub New(instanceName As String, tracking As List(Of Object))

MyBase.New(instanceName, tracking)

' Save the instance name as an unmanaged resource

umResource = Marshal.StringToCoTaskMemAuto(instanceName)

End Sub

Protected Overloads Overrides Sub Dispose(disposing As Boolean)

If disposed = False Then

If disposing Then

Console.WriteLine("[{0}].Derived.Dispose(true)", InstanceName)

' Release managed resources.

Else

Console.WriteLine("[{0}].Derived.Dispose(false)", InstanceName)

End If

' Release unmanaged resources.

If umResource <> IntPtr.Zero

Marshal.FreeCoTaskMem(umResource)

Console.WriteLine("[{0}] Unmanaged memory freed at {1:x16}", \_

InstanceName, umResource.ToInt64())

umResource = IntPtr.Zero

End If

disposed = True

End If

' Call Dispose in the base class.

MyBase.Dispose(disposing)

End Sub

' The derived class does not have a Finalize method

' or a Dispose method without parameters because it inherits

' them from the base class.

End Class

Public Class TestDisposal

Public Shared Sub Main()

Dim tracking As New List(Of Object)()

' Dispose is not called, Finalize will be called later.

Using Nothing

Console.WriteLine(vbNewLine + "Disposal Scenario: #1" + vbNewLine)

Dim d3 As New Derived("d1", tracking)

End Using

' Dispose is implicitly called in the scope of the using statement.

Using d1 As New Derived("d2", tracking)

Console.WriteLine(vbNewLine + "Disposal Scenario: #2" + vbNewLine)

End Using

' Dispose is explicitly called.

Using Nothing

Console.WriteLine(vbNewLine + "Disposal Scenario: #3" + vbNewLine)

Dim d2 As New Derived("d3", tracking)

d2.Dispose()

End Using

' Again, Dispose is not called, Finalize will be called later.

Using Nothing

Console.WriteLine(vbNewLine + "Disposal Scenario: #4" + vbNewLine)

Dim d4 As New Derived("d4", tracking)

End Using

' List the objects remaining to dispose.

Console.WriteLine(vbNewLine + "Objects remaining to dispose = {0:d}", tracking.Count)

For Each dd As Derived in tracking

Console.WriteLine(" Reference Object: {0:s}, {1:x16}",

dd.InstanceName, dd.GetHashCode())

Next dd

' Queued finalizers will be exeucted when Main() goes out of scope.

Console.WriteLine(vbNewLine + "Dequeueing finalizers...")

End Sub

End Class

' The program will display output similar to the following:

'

' Disposal Scenario: #1

'

'

' Disposal Scenario: #2

'

'

' [d2].Base.Dispose()

' [d2].Derived.Dispose(true)

' [d2] Unmanaged memory freed at 00000000001ce420

' [d2].Base.Dispose(true)

' [d2] Removed from tracking list: 0000000002bf8098

'

' Disposal Scenario: #3

'

'

' [d3].Base.Dispose()

' [d3].Derived.Dispose(true)

' [d3] Unmanaged memory freed at 00000000001ce420

' [d3].Base.Dispose(true)

' [d3] Removed from tracking list: 0000000000bb8560

'

' Disposal Scenario: #4

'

'

' Objects remaining to dispose = 2

' Reference Object: d1, 000000000297b065

' Reference Object: d4, 0000000003553390

'

' Dequeueing finalizers...

'

' [d4].Base.Finalize()

' [d4].Derived.Dispose(false)

' [d4] Unmanaged memory freed at 00000000001ce420

' [d4].Base.Dispose(false)

'

' [d1].Base.Finalize()

' [d1].Derived.Dispose(false)

' [d1] Unmanaged memory freed at 00000000001ce3f0

' [d1].Base.Dispose(false)

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl11_code');" \o "Copy Code)

using System;

using System.Collections.Generic;

using System.Runtime.InteropServices;

// Design pattern for a base class.

public abstract class Base : IDisposable

{

private bool disposed = false;

private string instanceName;

private List<object> trackingList;

public Base(string instanceName, List<object> tracking)

{

this.instanceName = instanceName;

trackingList = tracking;

trackingList.Add(this);

}

public string InstanceName

{

get

{

return instanceName;

}

}

//Implement IDisposable.

public void Dispose()

{

Console.WriteLine("\n[{0}].Base.Dispose()", instanceName);

Dispose(true);

GC.SuppressFinalize(this);

}

protected virtual void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

// Free other state (managed objects).

Console.WriteLine("[{0}].Base.Dispose(true)", instanceName);

trackingList.Remove(this);

Console.WriteLine("[{0}] Removed from tracking list: {1:x16}",

instanceName, this.GetHashCode());

}

else

{

Console.WriteLine("[{0}].Base.Dispose(false)", instanceName);

}

disposed = true;

}

}

// Use C# destructor syntax for finalization code.

~Base()

{

// Simply call Dispose(false).

Console.WriteLine("\n[{0}].Base.Finalize()", instanceName);

Dispose(false);

}

}

// Design pattern for a derived class.

public class Derived : Base

{

private bool disposed = false;

private IntPtr umResource;

public Derived(string instanceName, List<object> tracking) :

base(instanceName, tracking)

{

// Save the instance name as an unmanaged resource

umResource = Marshal.StringToCoTaskMemAuto(instanceName);

}

protected override void Dispose(bool disposing)

{

if (!disposed)

{

if (disposing)

{

Console.WriteLine("[{0}].Derived.Dispose(true)", InstanceName);

// Release managed resources.

}

else

{

Console.WriteLine("[{0}].Derived.Dispose(false)", InstanceName);

}

// Release unmanaged resources.

if (umResource != IntPtr.Zero)

{

Marshal.FreeCoTaskMem(umResource);

Console.WriteLine("[{0}] Unmanaged memory freed at {1:x16}",

InstanceName, umResource.ToInt64());

umResource = IntPtr.Zero;

}

disposed = true;

}

// Call Dispose in the base class.

base.Dispose(disposing);

}

// The derived class does not have a Finalize method

// or a Dispose method without parameters because it inherits

// them from the base class.

}

public class TestDisposal

{

public static void Main()

{

List<object> tracking = new List<object>();

// Dispose is not called, Finalize will be called later.

using (null)

{

Console.WriteLine("\nDisposal Scenario: #1\n");

Derived d3 = new Derived("d1", tracking);

}

// Dispose is implicitly called in the scope of the using statement.

using (Derived d1 = new Derived("d2", tracking))

{

Console.WriteLine("\nDisposal Scenario: #2\n");

}

// Dispose is explicitly called.

using (null)

{

Console.WriteLine("\nDisposal Scenario: #3\n");

Derived d2 = new Derived("d3", tracking);

d2.Dispose();

}

// Again, Dispose is not called, Finalize will be called later.

using (null)

{

Console.WriteLine("\nDisposal Scenario: #4\n");

Derived d4 = new Derived("d4", tracking);

}

// List the objects remaining to dispose.

Console.WriteLine("\nObjects remaining to dispose = {0:d}", tracking.Count);

foreach (Derived dd in tracking)

{

Console.WriteLine(" Reference Object: {0:s}, {1:x16}",

dd.InstanceName, dd.GetHashCode());

}

// Queued finalizers will be exeucted when Main() goes out of scope.

Console.WriteLine("\nDequeueing finalizers...");

}

}

// The program will display output similar to the following:

//

// Disposal Scenario: #1

//

//

// Disposal Scenario: #2

//

//

// [d2].Base.Dispose()

// [d2].Derived.Dispose(true)

// [d2] Unmanaged memory freed at 000000000034e420

// [d2].Base.Dispose(true)

// [d2] Removed from tracking list: 0000000002bf8098

//

// Disposal Scenario: #3

//

//

// [d3].Base.Dispose()

// [d3].Derived.Dispose(true)

// [d3] Unmanaged memory freed at 000000000034e420

// [d3].Base.Dispose(true)

// [d3] Removed from tracking list: 0000000000bb8560

//

// Disposal Scenario: #4

//

//

// Objects remaining to dispose = 2

// Reference Object: d1, 000000000297b065

// Reference Object: d4, 0000000003553390

//

// Dequeueing finalizers...

//

// [d4].Base.Finalize()

// [d4].Derived.Dispose(false)

// [d4] Unmanaged memory freed at 000000000034e420

// [d4].Base.Dispose(false)

//

// [d1].Base.Finalize()

// [d1].Derived.Dispose(false)

// [d1] Unmanaged memory freed at 000000000034e3f0

// [d1].Base.Dispose(false)

For an additional code example illustrating the design pattern for implementing **Finalize** and **Dispose**, see [Implementing a Dispose Method](http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx).

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifCustomizing a Dispose Method Name

Occasionally a domain-specific name is more appropriate than **Dispose**. For example, a file encapsulation might want to use the method name **Close**. In this case, implement **Dispose** privately and create a public **Close** method that calls **Dispose**. The following code example illustrates this pattern. You can replace **Close** with a method name appropriate to your domain. This example requires the [System](http://msdn.microsoft.com/en-us/library/system.aspx) namespace.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl41_ctl00_ctl01_code');" \o "Copy Code)

' Do not make this method overridable.

' A derived class should not be allowed

' to override this method.

Public Sub Close()

' Call the Dispose method with no parameters.

Dispose()

End Sub

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl41_ctl00_ctl02_code');" \o "Copy Code)

// Do not make this method virtual.

// A derived class should not be allowed

// to override this method.

public void Close()

{

// Call the Dispose method with no parameters.

Dispose();

}

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFinalize

The following rules outline the usage guidelines for the **Finalize** method:

* Implement **Finalize** only on objects that require finalization. There are performance costs associated with **Finalize** methods.
* If you require a **Finalize** method, consider implementing **IDisposable** to allow users of your class to avoid the cost of invoking the **Finalize** method.
* Do not make the **Finalize** method more visible. It should be **protected**, not **public**.
* An object's **Finalize** method should free any external resources that the object owns. Moreover, a **Finalize** method should release only resources that the object has held onto. The **Finalize** method should not reference any other objects.
* Do not directly call a **Finalize** method on an object other than the object's base class. This is not a valid operation in the C# programming language.
* Call the **base class's Finalize** method from an object's **Finalize** method.

|  |
| --- |
| **Description: NoteNote** |
| The base class's **Finalize** method is called automatically with the C# and C++ destructor syntax. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifDispose

The following rules outline the usage guidelines for the **Dispose** method:

* Implement the dispose design pattern on a type that encapsulates resources that explicitly need to be freed. Users can free external resources by calling the public **Dispose** method.
* Implement the dispose design pattern on a base type that commonly has derived types that hold onto resources, even if the base type does not. If the base type has a **Close** method, often this indicates the need to implement **Dispose**. In such cases, do not implement a **Finalize** method on the base type. **Finalize** should be implemented in any derived types that introduce resources that require cleanup.
* Free any disposable resources a type owns in its **Dispose** method.
* After **Dispose** has been called on an instance, prevent the **Finalize** method from running by calling the [GC.SuppressFinalize](http://msdn.microsoft.com/en-us/library/system.gc.suppressfinalize.aspx). The exception to this rule is the rare situation in which work must be done in **Finalize** that is not covered by **Dispose**.
* Call the base class's **Dispose** method if it implements **IDisposable**.
* Do not assume that **Dispose** will be called. Unmanaged resources owned by a type should also be released in a **Finalize** method in the event that **Dispose** is not called.
* Throw an **ObjectDisposedException** from instance methods on this type (other than **Dispose**) when resources are already disposed. This rule does not apply to the **Dispose** method because it should be callable multiple times without throwing an exception.
* Propagate the calls to **Dispose** through the hierarchy of base types. The **Dispose** method should free all resources held by this object and any object owned by this object. For example, you can create an object such as a **TextReader** that holds onto a **Stream** and an **Encoding**, both of which are created by the **TextReader** without the user's knowledge. Furthermore, both the **Stream** and the **Encoding** can acquire external resources. When you call the **Dispose** method on the **TextReader**, it should in turn call **Dispose** on the **Stream** and the **Encoding**, causing them to release their external resources.
* Consider not allowing an object to be usable after its **Dispose** method has been called. Re-creating an object that has already been disposed is a difficult pattern to implement.
* Allow a **Dispose** method to be called more than once without throwing an exception. The method should do nothing after the first call.